概论

定义：
并查集是一种树型的数据结构，用于处理一些不相交集合的合并及查询问题（即所谓的并、查）。比如说，我们可以用并查集来判断一个森林中有几棵树、某个节点是否属于某棵树等。

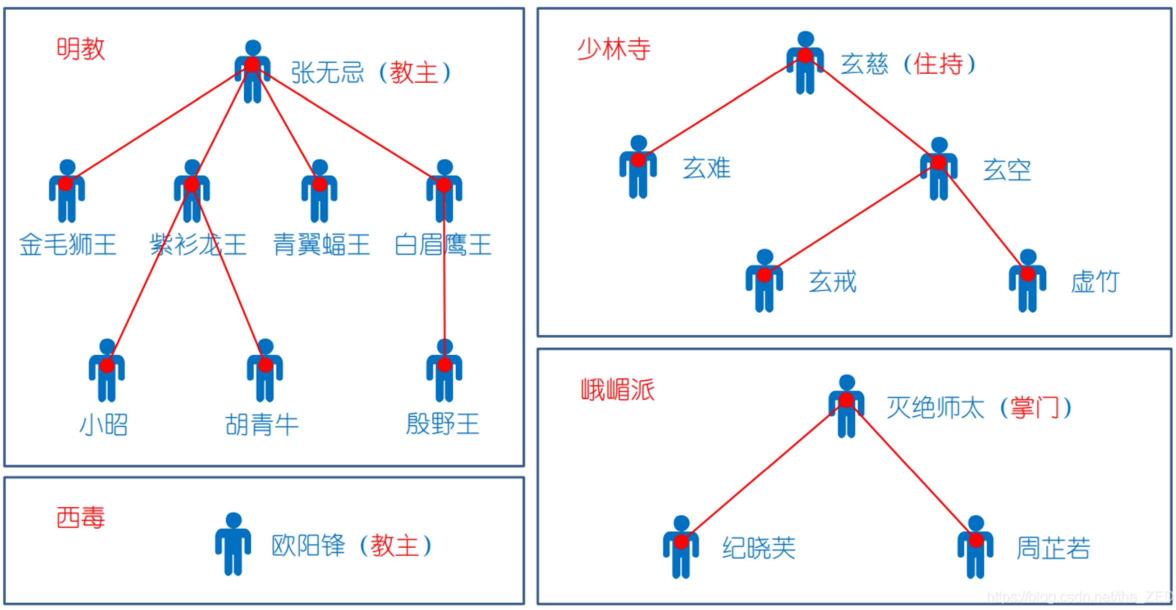
主要构成：
并查集主要由一个整型数组pre[ ]和两个函数find( )、join( )构成。
数组 pre[ ] 记录了每个点的前驱节点是谁，函数 find(x) 用于查找指定节点 x 属于哪个集合，函数 join(x,y) 用于合并两个节点 x 和 y 。

作用：
并查集的主要作用是求连通分支数（如果一个图中所有点都存在可达关系（直接或间接相连），则此图的连通分支数为1；如果此图有两大子图各自全部可达，则此图的连通分支数为2……）

并查集的现实意义

故事引入：
话说在江湖中散落着各式各样的大侠，他们怀揣着各自的理想和信仰在江湖中奔波。或是追求武林至尊，或是远离红尘，或是居庙堂之高，或是处江湖之远。尽管大多数人都安分地在做自己，但总有些人会因为彼此的信仰不同而聚众斗殴。因此，江湖上常年乱作一团，纷纷扰扰。
这样长期的混战，难免会打错人，说不定一刀就把拥有和自己相同信仰的队友给杀了。这该如何是好呢？于是，那些有着相同信仰的人们便聚在一起，进而形成了各种各样的门派，比如我们所熟知的“华山派”、“峨嵋派”、“,崆峒派”、“少林寺”、“明教”……这样一来，那些有着相同信仰的人们便聚在一起成为了朋友。以后再遇到要打架的事时，就不会打错人了。

但是新的问题又来了，原本互不相识的两个人如何辨别是否共属同一门派呢？
这好办！我们可以先在门派中选举一个“大哥”作为话事人（也就是掌门人，或称教主等）。这样一来，每当要打架的时候，决斗双方先自报家门，说出自己所在门派的教主名称，如果名称相同，就说明是自己人，就不必自相残杀了，否则才能进行决斗。于是，教主下令将整个门派划分为三六九等，使得整个门派内部形成一个严格的等级制度（即树形结构）。教主就是根节点，下面分别是二级、三级、……、N级队员。每个人只需要记住自己的上级名称，以后遇到需要辨别敌友的情况时，只需要一层层往上询问（网上询问）就能知道是否是同道中人了。



数据结构的角度来看：
由于我们的重点是在关注两个人是否连通，因此他们具体是如何连通的，内部结构是怎样的，甚至根节点是哪个（即教主是谁），都不重要。所以并查集在初始化时，教主可以随意选择（就不必再搞什么武林大会了），只要能分清敌友关系就行。

备注：上面所说的“教主”在教材中被称为“代表元”。
即：用集合中的某个元素来代表这个集合，则该元素称为此集合的代表元。

find( )函数的定义与实现

故事引入：
子夜，小昭于骊山下快马送信，发现一头戴竹笠之人立于前方，其形似黑蝠，倒挂于树前，甚惧，正系拔剑之时，只听四周悠悠传来：“如此夜深，姑凉竟敢擅闯明教，何不下坐陪我喝上一盅？”。小昭听闻，后觉此人乃明教四大护法之一的青翼蝠王韦一笑，答道：“在下小昭，乃紫衫龙王之女”。蝠王轻惕，急问道：“尔等既为龙王之女，故同为明教中人。敢问阁下教主大名，若非本教中人，于明教之地肆意走动那可是死罪！”。小昭吓得赶紧打了个电话问龙王：“龙王啊，咱教主叫啥名字来着？”，龙王答道：“吾教主乃张无忌也！”，小昭遂答蝠王：“张无忌！”。蝠王听后，抱拳请礼以让之。
在上面的情境中，小昭向他的上级（紫衫龙王）请示教主名称，龙王在得到申请后也向他的上级（张无忌）请示教主名称，此时由于张无忌就是教主，因此他直接反馈给龙王教主名称是“张无忌”。同理，青翼蝠王也经历了这一请示过程。
在并查集中，用于查询各自的教主名字的函数就是我们的find()函数。find(x)的作用是用于查找某个人所在门派的教主，换言之就是用于对某个给定的点x，返回其所属集合的代表。

实现：
首先我们需要定义一个数组：int pre[1000]; （数组长度依题意而定）。这个数组记录了每个人的上级是谁。这些人从0或1开始编号（依题意而定）。比如说pre[16]=6就表示16号的上级是6号。如果一个人的上级就是他自己，那说明他就是教主了，查找到此结束。也有孤家寡人自成一派的，比如欧阳锋，那么他的上级就是他自己。
每个人都只认自己的上级。比如小昭只知道自己的上级是紫衫龙王。教主是谁？不认识！要想知道自己教主的名称，只能一级级查上去。因此你可以视find(x)这个函数就是找教主用的。
下面给出这个函数的具体实现：

```
int find(int x)                 //查找x的教主{
    while(pre[x] != x)          //如果x的上级不是自己（则说明找到的人不是教主）
        x = pre[x];             //x继续找他的上级，直到找到教主为止
    return x;                   //教主驾到~~~
}
```

join( )函数的定义与实现

故事引入：
虚竹和周芷若是我个人非常喜欢的两个人物，他们的教主分别是玄慈方丈和灭绝师太，但是显然这两个人属于不同门派，但是我又不想看到他们打架。于是，我就去问了下玄慈和灭绝：“你看你们俩都没头发，要不就做朋友吧”。他们俩看在我的面子上同意了，这一同意非同小可，它直接换来了少林和峨眉的永世和平。

实现：
在上面的情景中，两个已存的不同门派就这样完成了合并。这么重大的变化，要如何实现？要改动多少地方？其实很简单，我对玄慈方丈说：“大师，麻烦你把你的上级改为灭绝师太吧。这样一来，两派原先所有人员的教主就都变成了师太，于是下面的人们也就不会打起来了！反正我们关心的只是连通性，门派内部的结构不要紧的”。玄慈听后立刻就不乐意了：“我靠，凭什么是我变成她手下呀，怎么不反过

来？我抗议！”。抗议无效，我安排的，最大。反正谁加入谁效果是一样的，我就随手指定了一个，join()函数的作用就是用来实现这个的。

join(x,y)的执行逻辑如下：
1、寻找 x 的代表元（即教主）；
2、寻找 y 的代表元（即教主）；
3、如果 x 和 y 不相等，则随便选一个人作为另一个人的上级，如此一来就完成了 x 和 y 的合并。
下面给出这个函数的具体实现：

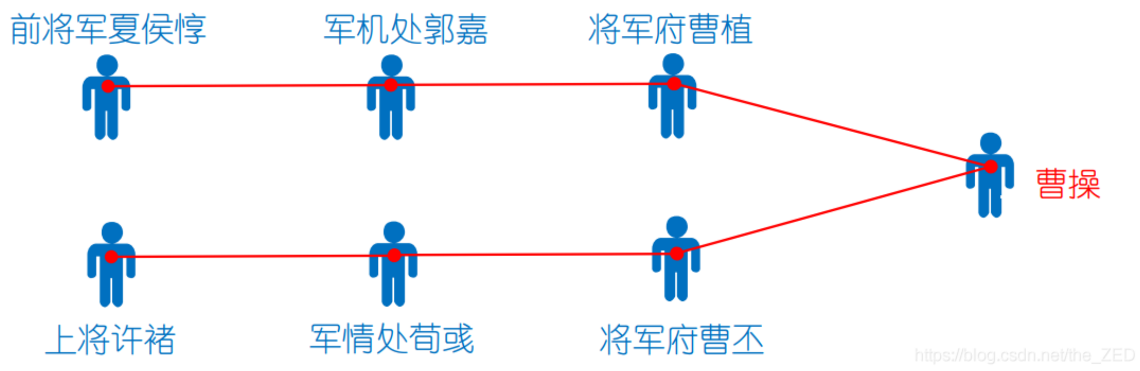
```
void join(int x,int y)                    //我想让虚竹和周芷若做朋友
{
    int fx=find(x), fy=find(y);           //虚竹的老大是玄慈，芷若MM的老大是灭绝
    if(fx != fy)                          //玄慈和灭绝显然不是同一个人
        pre[fx]=fy;                       //方丈只好委委屈屈地当了师太的手下啦
}
```

路径压缩算法之一（优化find( )函数）

问题引入：
前面介绍的 join(x,y) 实际上为我们提供了一个将不同节点进行合并的方法。通常情况下，我们可以结合着循环来将给定的大量数据合并成为若干个更大的集合（即并查集）。但是问题也随之产生，我们来看这段代码：

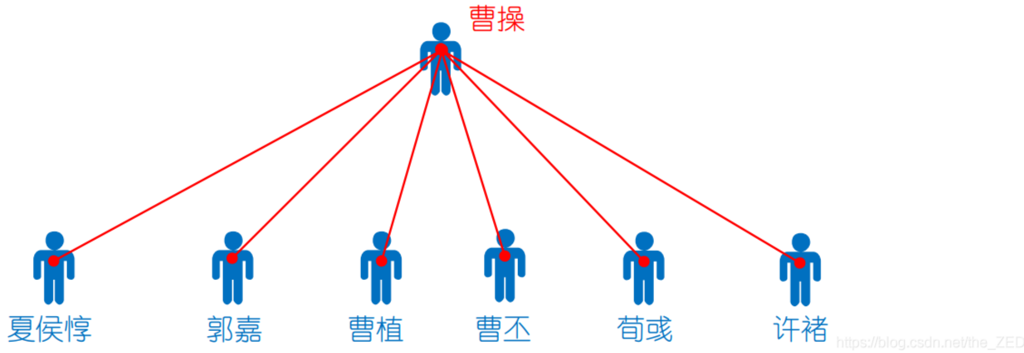
```
if(fx != fy)
    pre[fx]=fy;
```

这里并没有明确谁是谁的前驱（上级）的规则，而是我直接指定后面的数据作为前面数据的前驱（上级）。那么这样就导致了最终的树状结构无法预计，即有可能是良好的 n 叉树，也有可能是单支树结构（一字长蛇形）。试想，如果最后真的形成单支树结构，那么它的效率就会及其低下（树的深度过深，那么查询过程就必然耗时）。
而我们最理想的情况就是所有人的直接上级都是教主，这样一来整个树的结构就只有两级，此时查询教主只需要一次。因此，这就产生了路径压缩算法。
设想这样一个场景：两个互不相识的大将夏侯惇和许褚碰面了，他们都互相看不惯，想揍对方。于是按照江湖规矩，先打电话问自己的上级：“你是不是教主？”上级说：“我不是呀，我的上级是***，我帮你问一下。”就这样两个人一路问下去，直到最终发现他们的教主都是曹操。具体结构如下：



“失礼失礼，原来是自己人，在下军机处前将军夏侯惇!”
“幸会幸会，不打不相识嘛，在下军情处上将许褚！”
紧接着，两人便高高兴兴地手拉手喝酒去了。
“等等等等，两位同学请留步，还有事情没完成呢！”我叫住他俩：“还要做路径压缩！”
两人醒悟，夏侯惇赶紧打电话给他的上级郭嘉：“军师啊，我查过了，我们的教主都是曹丞相。不如我们直接拜在丞相手下吧，省得级别太低，以后找起来太麻烦。”郭嘉答道：“所言甚是”。

许褚接着也打电话给刚才拜访过的荀彧，做了同样的事。于是此时，整个曹操阵营的结构如下所示：



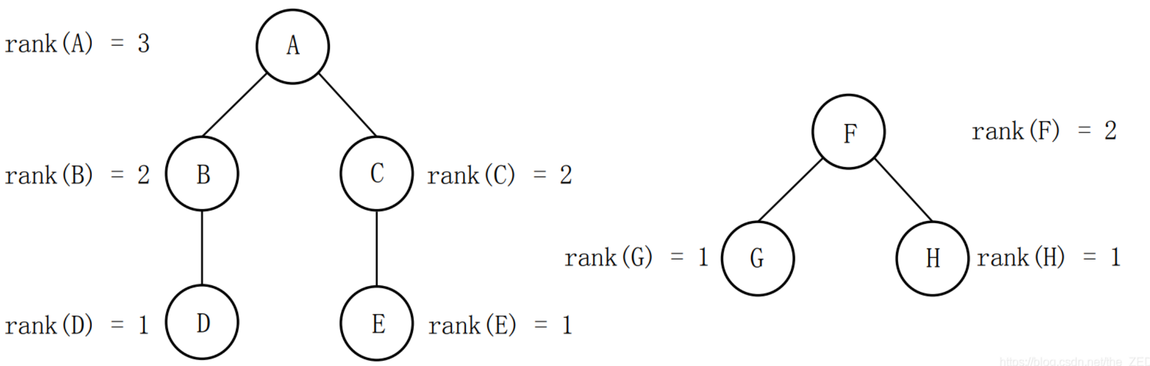
实现：
从上面的查询过程中不难看出，当从某个节点出发去寻找它的根节点时，我们会途径一系列的节点（比如上面的例子中，从夏侯惇→郭嘉→曹植→曹操），在这些节点中，除了根节点外（即曹操），其余所有节点（即夏侯惇、郭嘉、曹植）都需要更改直接前驱为曹操。
因此，基于这样的思路，我们可以通过递归的方法来逐层修改返回时的某个节点的直接前驱（即pre[x]的值）。简单说来就是将x到根节点路径上的所有点的pre（上级）都设为根节点。下面给出具体的实现代码：

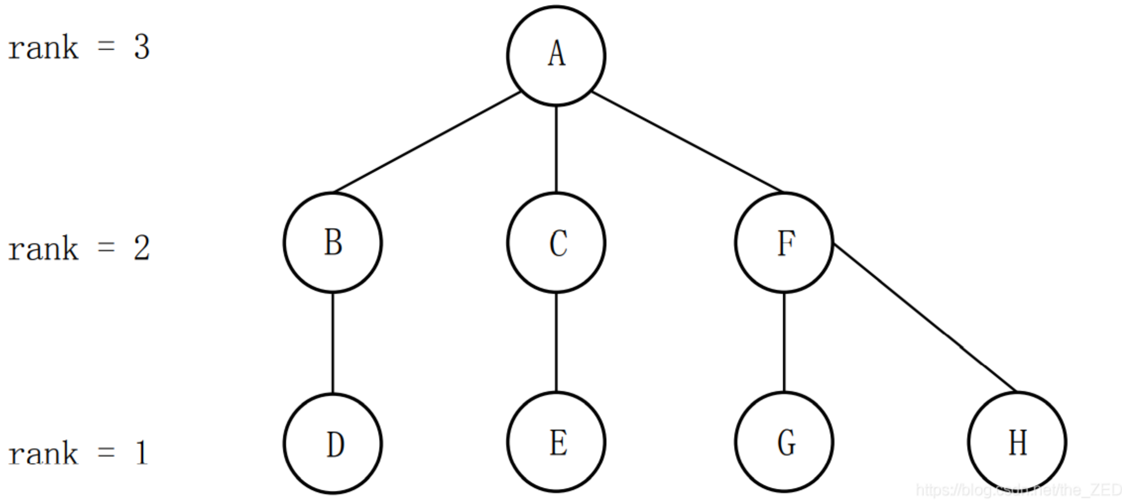
```
int find(int x)                    //查找结点 x的根结点 {
    if(pre[x] == x) return x;      //递归出口：x的上级为 x本身，即 x为根结点
    return pre[x] = find(pre[x]);  //此代码相当于先找到根结点 rootx，然后pre[x]=rootx
}
```

路径压缩算法之二（加权合并）

备注：
不要一看到"加权标记"这种比较高大上的名词就害怕，其实他也是属于路径压缩算法，不同的是其思想更加高级（更不容易想到）罢了。不过我还是说一下，掌握了前面几点内容就能应对绝大多数ACM问题，下面的"加权标记法"供有兴趣的同学学习交流。

主要思路：
加权标记法需要将树中所有节点都增设一个权值，用以表示该节点所在树中的高度（比如用rank[x]=3表示 x 节点所在树的高度为3）。这样一来，在合并操作的时候就能通过这个权值的大小来决定谁当谁的上级（玄慈哭了："正义终会来到，但永不会缺席"）。
在合并操作的时候，假设需要合并的两个集合的代表元（教主）分别为 x 和 y，则只需要令pre[x] = y 或者pre[y] = x 即可。但我们为了使合并后的树不产生退化（即：使树中左右子树的深度差尽可能小），那么对于每一个元素 x，增设一个rank[x]数组，用以表达子树 x 的高度。在合并时，如果rank[x] < rank[y]，则令pre[x] = y；否则令pre[y] = x。
举个例子，我们对以A，F为代表元的集合进行合并操作（如下图所示）：





可以看出，合并前两个树的最大高度为3，合并后依然是3，这也就达到了我们的目的。但如果令pre[A]=F，那么就会使得合并后的树的总高度增加，这里我就不上图了，同学们不信可以自己画出来看看。
我们常说，鱼和熊掌不可兼得，同理，时间复杂度和空间复杂度也很难兼得。由于给每个节点都增加了一个权值来标记其在树中的高度，这也就意味着需要额外的数据结构来存放权重信息，所以这将导致额外的空间开销。

实现：
加权标记法的核心在于对rank数组的逻辑控制，其主要的情况有：
1、如果rank[x] < rank[y]，则令pre[x] = y；
2、如果rank[x] == rank[y]，则可任意指定上级；
3、如果rank[x] > rank[y]，则令pre[y] = x；
在实际写代码时，为了使代码尽可能简洁，我们可以将第1点单独作为一个逻辑选择，然后将2、3点作为另一个选择（反正第2点任意指定上级嘛），所以具体的代码如下：

```
void union(int x,int y){
    x=find(x);                          //寻找 x的代表元
    y=find(y);                          //寻找 y的代表元
    if(x==y) return ;                   //如果 x和 y的代表元一致，说明他们共属同一集合，
则不需要合并，直接返回；否则，执行下面的逻辑
    if(rank[x]>rank[y]) pre[y]=x;       //如果 x的高度大于 y，则令 y的上级为 x
    else                                //否则
    {
        if(rank[x]==rank[y]) rank[y]++; //如果 x的高度和 y的高度相同，则令 y的高度加1
        pre[x]=y;                       //让 x的上级为 y
    }
}
```

权值并查集

顾名思义，通过权值来设置特殊性质的并查集。

洛谷 / 题目列表 / 题目详情

P2024 [NOI2001] 食物链

提交答案 加入题单 复制题目

题目描述

复制Markdown 展开

动物王国中有三类动物 A, B, C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。

现有 N 个动物，以 $1 \sim N$ 编号。每个动物都是 A, B, C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 `1 X Y`，表示 X 和 Y 是同类。
- 第二种说法是 `2 X Y`，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N 和 K 句话，输出假话的总数。

输入格式

第一行两个整数，N, K，表示有 N 个动物，K 句话。

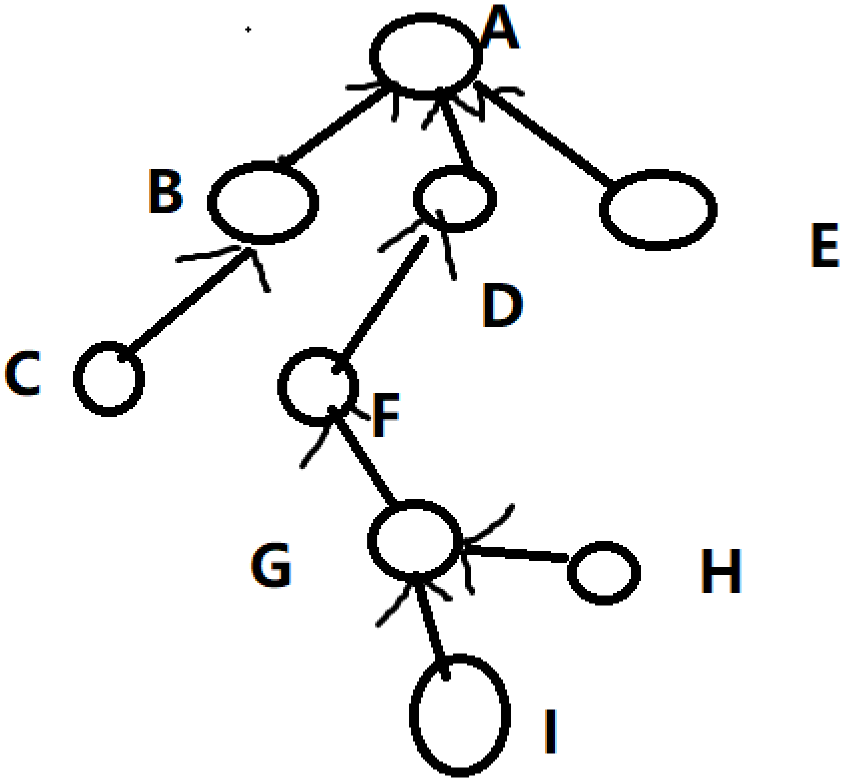
第二行开始每行一句话（按照题目要求，见样例）

输出格式

一行，一个整数，表示假话的总数。

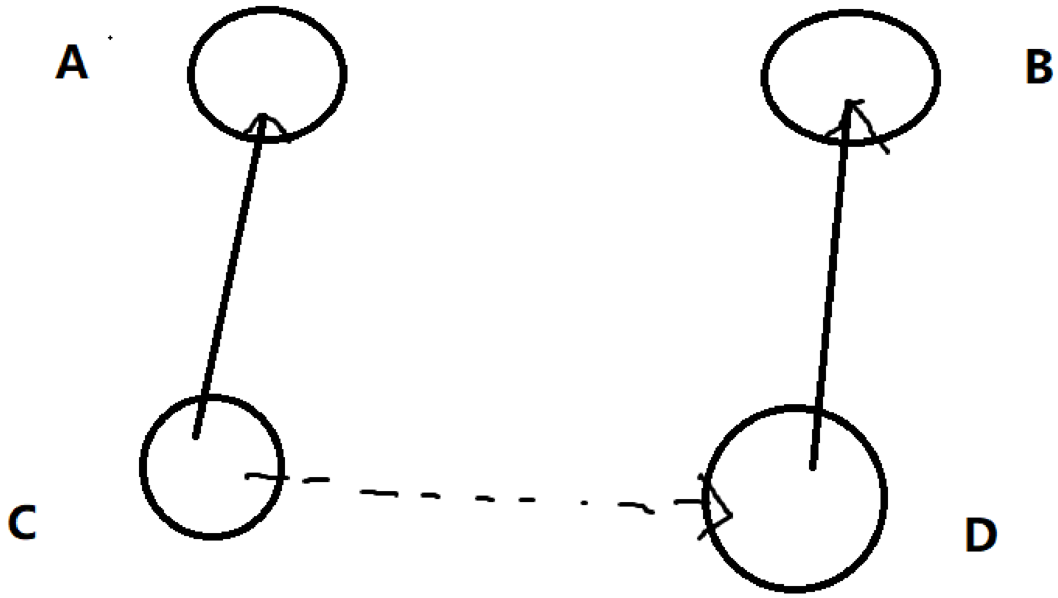
咋们通过以上题目来进行讲解。

我们思考A -> B B-> C 的话 一定是能推出 C->A，所以ABC在一个集合里面，我们只需要知道其中两个的关系，就能推导第三个。



我们假设这是一些动物的关系，我们可以通过图片看出来，A被B,D,E吃，C,F被A吃，G是A的同类。

那么我们发现，若我们当前动物层数是X，那么与我们为同类的即为，X + 3 * n层，吃我们的是 X + 1 + 3 * n层，被我们吃的是 X + 2 + 3 * n层
对于3取模的话就只剩3种情况，0，1，2。所以最后，我们只需要维护每个点到他祖宗的深度就可以了。
那么主要难点的话还是合并，我们该怎么合并呢？



假设C，D同类我们现在要合并他们的祖宗节点A，C
由于 AB + AC == BD + CD 所以 AC + AB ≡ BD (mod 3)
也就是说 (AC + AB - BD) mod 3 == 0
所以AB = BD - AC

```
//这是求深度的关键代码
int find(int x){
    if (p[x] != x){
        int t = find(p[x]);
        d[x] += d[p[x]];
        p[x] = t;
    }
    return p[x];
}
```

总体食物链代码

```
#include <iostream>

using namespace std;

const int N = 50010;

int n, m;
int p[N], d[N];

int find(int x){
    if (p[x] != x){
        int t = find(p[x]);
        d[x] += d[p[x]];
        p[x] = t;
    }
    return p[x];
}

int main(){
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) p[i] = i;
    int res = 0;
    while (m -- ){
        int t, x, y;
        scanf("%d%d%d", &t, &x, &y);
        if (x > n || y > n) res ++ ;
        else{
            int px = find(x), py = find(y);
            if (t == 1){
                if (px == py && (d[x] - d[y]) % 3) res ++ ;
                else if (px != py){
                    p[px] = py;
                    d[px] = d[y] - d[x];
                }
            }else{
                if (px == py && (d[x] - d[y] - 1) % 3) res ++ ;
                else if (px != py){
                    p[px] = py;
                    d[px] = d[y] + 1 - d[x];
                }
            }
        }
    }
    printf("%d\n", res);
```

```
    return 0;
}
```